UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Parallelisation of merging of paired-end reads from Illumina sequencing using GPUs

Torgeir Pilskog Tynes

Master's Thesis
Autumn 2018

Parallelisation of merging of paired-end reads from Illumina sequencing using GPUs

Torgeir Pilskog Tynes

August 2, 2018

Contents

List of Figures

List of Tables

Abstract

Sequencing projects are using paired-end reads to compensate for the relatively short length of the reads obtained by some sequencing technologies such as Illumina. To take advantage of the resulting paired-end reads, we need software that can handle paired-end reads by merging their overlapping parts. There is an ever-growing plethora of these software tools, all trying to be a little better in their respective fields to improve the result. However, none of these tools employ parallelisation on Graphical Processing Units(GPUs) to speed up the merging of the paired-end reads. What we aim to achieve in this project is a respectable speedup compared to existing solutions when it comes to merging of paired-end reads. The speedup might also give an opportunity to make better calculations of the score to achieve a better result.

For this, I have developed a GPU implementation of a paired-end read merger based on FLASH, which employs parallelization on the GPU. This implementation was tested against FLASH in both speed and accuracy. Although the GPU implementation cannot quite catch up to FLASH, simple optimizations would allow it to easily compete with it, and shows great potential for paired-end read merging on GPUs.

Acknowledgements

First and foremost I would like to thank my supervisor, Torbjørn Rognes, for all the help and guidance throughout this master's thesis.

I would also like to thank my friends and family for all the support and encouragement they have provided me.

1 Introduction

1.1 Motivation

Sequencing projects are using paired-end reads to compensate for the relatively short length of the reads obtained by some sequencing technologies such as Illumina. To take advantage of the resulting paired-end reads, we need software that can handle paired-end reads by merging their overlapping parts. There is an ever-growing plethora of these software tools, all trying to be a little better in their respective fields to improve the result. However, none of these tools employ parallelisation on Graphical Processing Units(GPUs) to speed up the merging of the paired-end reads. What we aim to achieve in this project is a respectable speedup compared to existing solutions when it comes to merging of paired-end reads. The speedup might also give an opportunity to make better calculations of the score to achieve a better result.

1.2 Problem Definition

1.2.1 Merging and Scoring

Finding the optimal score for each of the paired-end reads is the final goal here. This is done by aligning the two overlapping regions in the reads together by calculating the score of each base. The highest of these scores is thus the best alignment of the overlap and that is the one we keep. If the score is lower than a set threshold however, the reads are not to be merged.

1.2.2 Speed

Obviously, the more reads we have, the longer the merging will take. Most of the time is spent on trying each combination of the overlaps and calculating the score for each of them. Many of the current algorithms uses a simplified scoring algorithm to improve speed, sacrificing correctness of the overlaps in the merging.

1.2.3 Goal

To achieve the goal I will try to use parallelisation on GPUs.
The goal also includes:

- Implement and optimize a tool for merging paired-end reads on an NVIDIA Quadro P6000 GPU.
- Study whether a GPU is effective for doing parallelisation of merging of paired-end reads.
- Determine whether the finished tool is able to compete with other already existing tools of the same nature.

2 Background

2.1 DNA Sequencing

The DNA consists of bases of A(adenine), T(thymine), C(cytosine) and G(guanine).
With sequencing one can read these sequences of bases, and by studying them one can find its functions. To read these bases we use DNA sequencing. Sequencing is carried out by sequencing platforms such as Illumina. In Illumina sequencing we have three steps: amplifying, sequencing, and analyzing. We start with purified DNA and fragment it into smaller pieces, these fragments are then given adapters that act as reference points during the amplification, sequencing and analysis. The DNA is then loaded onto a special chip called a flow cell where the amplification and sequencing takes place. Anchored along the bottom of this chip we find hundreds of thousands of oligonucleotides. Oligonucleotides are short, synthetic fragments of DNA which can attach to complementary sequences. Once the fragments have attached to the oligonucleotides, the cluster generation phase begins. The cluster generation phase duplicates the DNA fragments, making thousands of copies. After this, primers and nucleotides enter the chip. The primers add one nucleotide at a time with fluorescent tags, and a camera takes a picture of the chip after each round of synthesis. Software then determines what base the color of each fluorescent tag on the chip corresponds to. What we then get out of the sequencing machine is millions or billions of reads of different parts of a genome. Later we will have to use software assisted tools to find which of these reads belong together and if they contain errors. The assembled DNA string we get out of this should be very similar to the actual DNA put into the machine(Illumina 2017a).

2.1.1 DNA Sequencing Errors

When doing sequencing, the process will always produce some errors in the resulting reads. In Illumina sequencing we have three possible sequencing errors: substitutions, insertions, and deletions. Of these three, substitutions are the most common.

1. Substitutions happen when one nucleotide is switched with another, i.e. an A becomes a T.

2. Deletions is when the read output is missing a nucleotide, i.e. the sequence $AGGCT$ is input, and the resulting sequencing shows $AGGT$, missing a C.

3. Insertions is when an extra nucleotide is added to a sequence, i.e. $GCCT$ is input, and the output shows $GCCAT$, where A is inserted.

2.2 Single-read Sequencing

Single-read sequencing is the simplest method of doing read sequencing. When doing single-read sequencing, the read is only sequenced from one end. This type of sequencing is usually both the simplest and the cheapest method(Illumina 2018). However, the resulting reads have a tendency to decrease in quality the further the read is sequenced. The quality drop is due to substitutions, deletions and insertions which become more frequent at the ends.

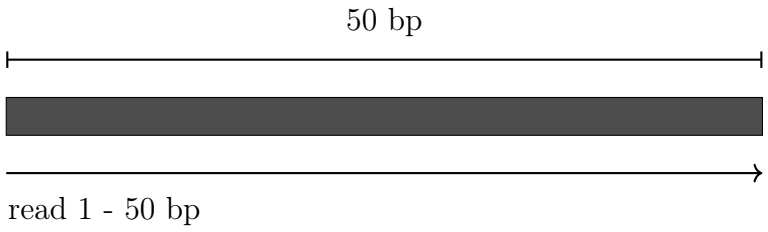

Figure 1: Representation of a single-end read

2.3 Paired-end Sequencing

Paired-end sequencing is a method of sequencing, so that you can achieve better results. It allows for longer reads and higher quality of the bases. It is achieved by the sequencer by reading a fragment from both directions where there is an overlap in the middle. Software then finds where these two pairs overlap best and combines the two into one sequence. This process is called merging. Paired-end sequencing can improve the quality by detecting substitutions, insertions or deletions in the overlapping regions. If there are differences in the overlapping regions, the nucleotide with the highest quality score can be chosen as the nucleotide in the final sequence. Normally you would have paired-end reads that overlap, but it is also possible to have paired-end reads which do not overlap. Non-overlapping paired-end reads allow for longer fragments, but no coverage between the two pairs, the distance between the two pairs are however known.

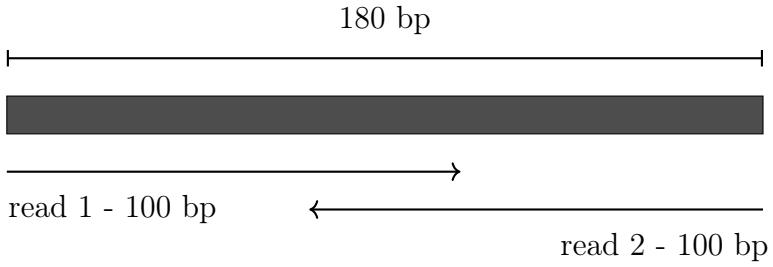

Figure 2: Representation of a paired-end read

2.4 Input Format

The input containing the sequences is given in a FASTQ file. FASTQ is a format that emerged from the simpler FASTA format. In addition to storing each nucleotide in a sequence, the FASTQ file format also stores the quality score for each of the nucleotides. Since there is no standard as to how sequences should be stored, several formats have appeared. As for FASTQ, it started with the original Sanger format, and later Solexa, Illumina and several others developed their own format(Cock et al. 2009). In this thesis we will focus on the Illumina FASTQ format.

Table 1: Elements in the sequence identifier

Element	**Requirements**	**Description**
@	@	Each sequence identifier line starts with @
<instrument>	Characters allowed: a-z, A-Z, 0-9 and underscore	Instrument ID
<run number>	Numerical	Run number on instrument
<flowcell ID>	Characters allowed: a-z, A-Z, 0-9	
<lane>	Numerical	Lane number
<tile>	Numerical	Tile number
<x_pos>	Numerical	X coordinate of cluster
<y_pos>	Numerical	Y coordinate of cluster
<read>	Numerical	Read number. 1 can be single read or read 2 of paired-end
<is filtered>	Y or N	Y if the read is filtered, N otherwise
<control number>	Numerical	0 when none of the control bits are on, otherwise it is an even number.
<index sequence>	ACTG	Index sequence

Each entry in a FASTQ file consists of four lines(Illumina 2017(b)):

1. Sequence identifier
2. Sequence
3. Quality score identifier line
4. Quality score

The sequence identifier needs to be in the following format:
`@<instrument>:<run number>:<flowcell ID>:<lane>:<tile>:<x-pos>:<y-pos>`
`<read>:<is filtered>:<control number>:<index sequence>`.

The elements are described in *Table 1* above.

An Illumina FASTQ file can look as follows:

```
@EAS139:136:FC706VJ:2:5:1000:12850 1:Y:18:ATCACG
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
+
BBBBCCCC?<A?BC?7@@???????DBBA@@@@A@@
```

When merging paired-end reads, we need two FASTQ files, one file for the forward reads, and one file for the reverse reads.

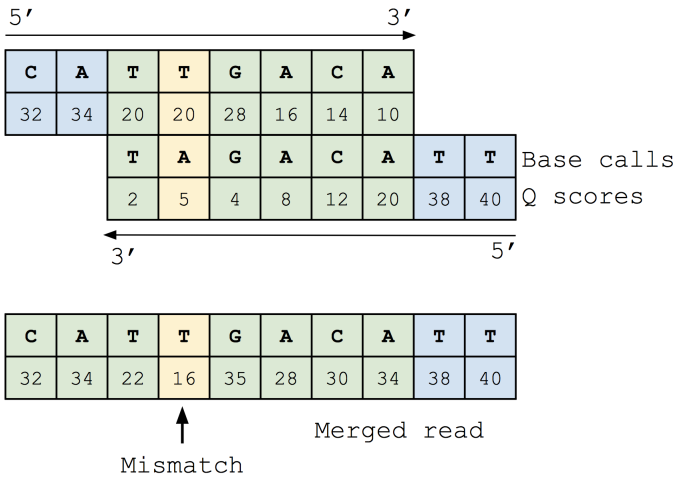

Figure 3: How the scoring and merging for paired-end reads is determined(Robert C. et al. 2015)

2.5 Scoring

Illumina gives a quality score for all of its bases in a sequence. This is called the Phred score, Phred score is an integer value representing the estimated probability of an error in a given base. The Phred score can however only say something about substitution errors as indels is not possible to detect at this stage. For any base call at position i, the estimated error probability is given by $Q_i = -10 \log_{10} p_i$, where Q is the Phred score of that base, and p is the error rate. To get the correct result, both the p and the resulting Phred score's ASCII value must be subtracted by 33. Today, typical base call error rates range from 0.1 to 10%(Robert C. et al. 2015). A quality score for a read as a whole is also needed, and is given by the sum of error probabilities:

$$E = \sum_i p_i = \sum_i 10^{-Q_i/10}$$

During the merging of the two fragments in paired-end sequencing, a score needs to be calculated for the newly merged read based on matches and mismatches from the forward and reverse reads. We also need a score threshold, if the quality score for the sequence is below the threshold, the reads will not be merged.

2.5.1 Merging

When two reads overlap, the score for each new base must be calculated. As Edgar et al. shows in their paper, if two reads agree on a base call the quality score increases per the following equation:

$$(p_x p_y/3)/(1 - p_x - p_y + 4p_x p_y/3)$$

Here x and y is the base call for the forward and the reverse read respectively.

If there is a mismatch, the base call with the highest quality is chosen, and

the new quality score is reduced per the following equation:

$$p_x(1 - p_y/3)/(p_x + p_y - 4p_xp_y/3)$$

This scoring scheme is shown in figure 3 above.

In this thesis we will use the FLASH(Magoc et al. 2011) algorithm, but we will change the scoring to the formulas given above, hopefully providing a more accurate score in the outputs. FLASH uses a simple scoring algorithm which calculates the score for an overlap as the ratio between the number of mismatches and the overlap length. FLASH does ungapped alignment, taking advantage of Illumina's low rate of gaps in the resulting reads.

2.6 General Algorithm in FLASH

The algorithm needed for the merging is stated in the paper about the FLASH algorithm(Magoc et al. 2011). To find the correct overlap it considers every possible legal overlap between the paired-end reads. A legal overlap is defined as an ungapped alignment between the reads where at least *min-olap* bases overlap. The algorithm goes as follows:

1. Align the pair of reads so that they overlap completely; e.g. by the full length of the shorter read.

2. Repeat while the overlap is longer than *min-olap*:

   (a) Calculate the overlap length. If an 'N' occurs in the overlapping region, it is not counted towards the overlap length.

   (b) Calculate the score for the overlap as the ratio between the number of mismatches and the overlap length, ignoring N's.

   (c) If the score of the overlap is smaller than the score of the best overlap, save it as the new best overlap.

   (d) If the score is equal to the best previous score:

      i. Calculate the average quality value of all mismatches in the overlap.

ii. If the average quality value is smaller than the average quality value of the mismatches in the best overlap, save the current overlap as the best overlap.

(e) Slide the reads apart by one base, reducing the overlap by one.

3. Compare the score of the best overlap to the mismatch threshold. If the score is bigger than the mismatch threshold, report that no good overlap was found, otherwise, return the best overlap.

We also have a maximum number of overlaps given by $O_{max} = R - m$, where R is the read length and m is the minimum overlap. Due to Illuminas increasing error rate towards the 3'-end of reads, the scoring scheme will prefer longer overlaps while this will not always be correct. To counter this and determine whether a long overlap is better than a shorter one, we define *max-olap* to be the maximum length of the overlap expected in 90% of read pairs for a given read length and fragment size. For the fragments that then exceeds the *max-olap*, we calculate their score as the ratio between the number of mismatches and *max-olap* rather than the overlap length. We also need a way to distinguish between low-quality overlapping reads and non-overlapping reads. For this we use *mr*, the maximum proportion of mismatches that we allow in the overlapping region. Usually, *mr* is set to 0.25. If the overlap found has a mismatch ratio less than or equal to *mr*, the reads are merged.

2.7 Graphical Processing Unit

Graphical Processing Units(GPUs) are used for manipulating computer graphics and image processing. They take use of highly parallel structures to make them more efficient than a CPU at algorithms which process large blocks of data in parallel.
We also have GPUs for high-performance computing such as the NVIDIA Tesla and Quadro products. These products are aimed towards scientific computing applications, such as Biomedical Informatics and Artificial Intelligence.
For this thesis we will be using an NVIDIA Quadro P6000 GPU(NVIDIA 2016), which is based on the GP102GL architecture. The GP102GL architecture features 30 streaming multiprocessors(SMs), with 3840 CUDA cores divided between them. Each of the streaming processors is massively threaded. The GP102GL architecture is shown below in *Figure 4*.

2.7.1 CUDA

CUDA is the parallel computing platform that allows a developer to easily use all the features of a CUDA-enabled GPU. Prior to CUDA one would have to be quite skilled in graphics programming in either Direct3D or OpenGL to make use of the computational powers of the GPUs. CUDA bypasses this by being a software layer that enables the use of C/C++ or Fortran to program the GPU, thus no knowledge in graphical programming necessary. When programming in CUDA, we have to look at it as a system which consists of a host(the CPU) and one or more devices(the GPUs). Since the host and the device is two different components data must be explicitly copied from the host to the device and back. On the device we have *global memory*, this memory tends to have long access latencies and finite bandwidth. That is why we also have on-chip memories: registers and shared memory per SM block, these have limited capacity, but are much faster. Registers are private for individual threads while the shared memory is accessible for all threads within a thread block.

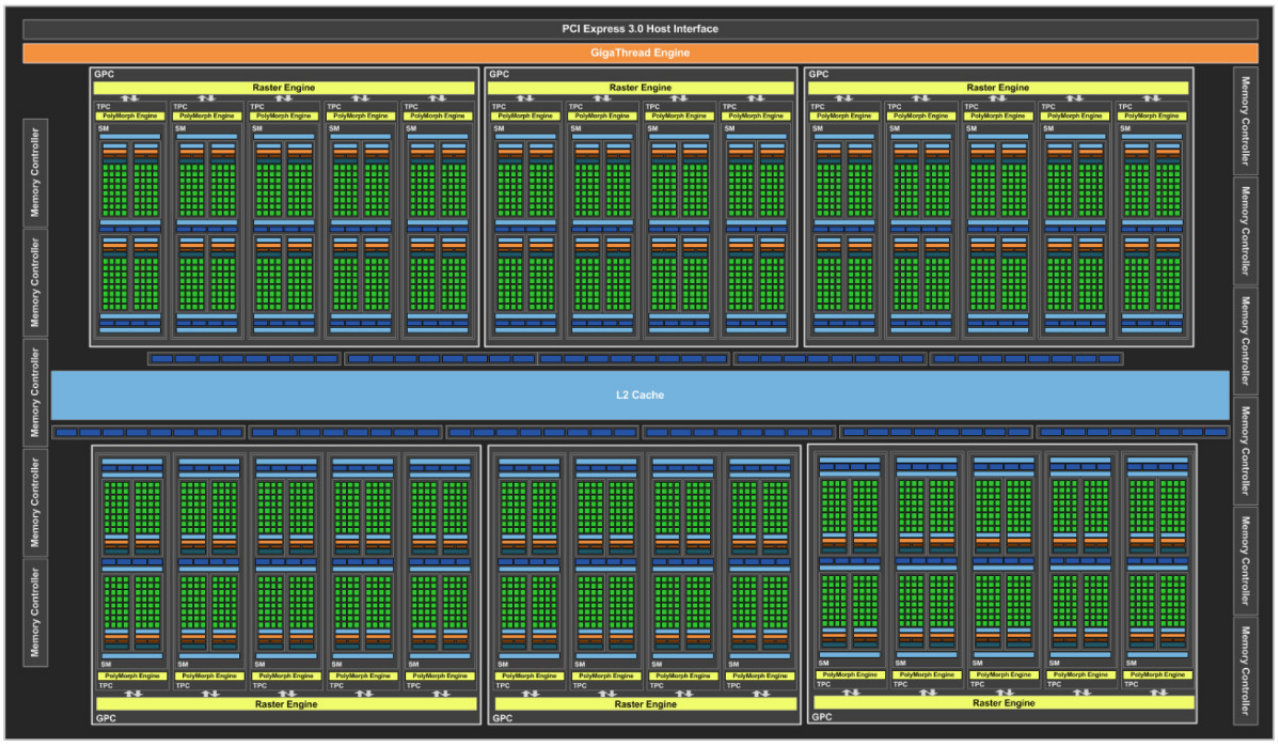

Figure 4: Pascal GP102GL Fill chip block diagram

2.7.2 Streaming Multiprocessors

The streaming multiprocessors or SMs are the fundamental part of a NVIDIA GPU. These SMs have the ability to execute hundreds of threads concurrently, and to handle and keep track of everything the SIMT(Single-Instruction, Multiple-Thread)(NVIDIA 2012) architecture is used. This architecture is built around hardware-multithreading instead of instruction-level parallelism.

The architecture of the multiprocessors are different on many of the NVIDIA GPUs, the Streaming Multiprocessors in the Pascal cards are shown in *Figure 5*.

Figure 5: Pascal Streaming Multiprocessor

In the Streaming Multiprocessor we find the *warp scheduler*. The *warp scheduler* creates, manages, schedules and executes parallel threads in *warps*. Each of the *warps* consists of 32 threads. When the *warp* is executed, the threads start out at the same program address, but they are free to branch and execute independently. When the *warp scheduler* receives one or more thread blocks to execute, these are partitioned into different warps. This is shown in *Figure 6* below.

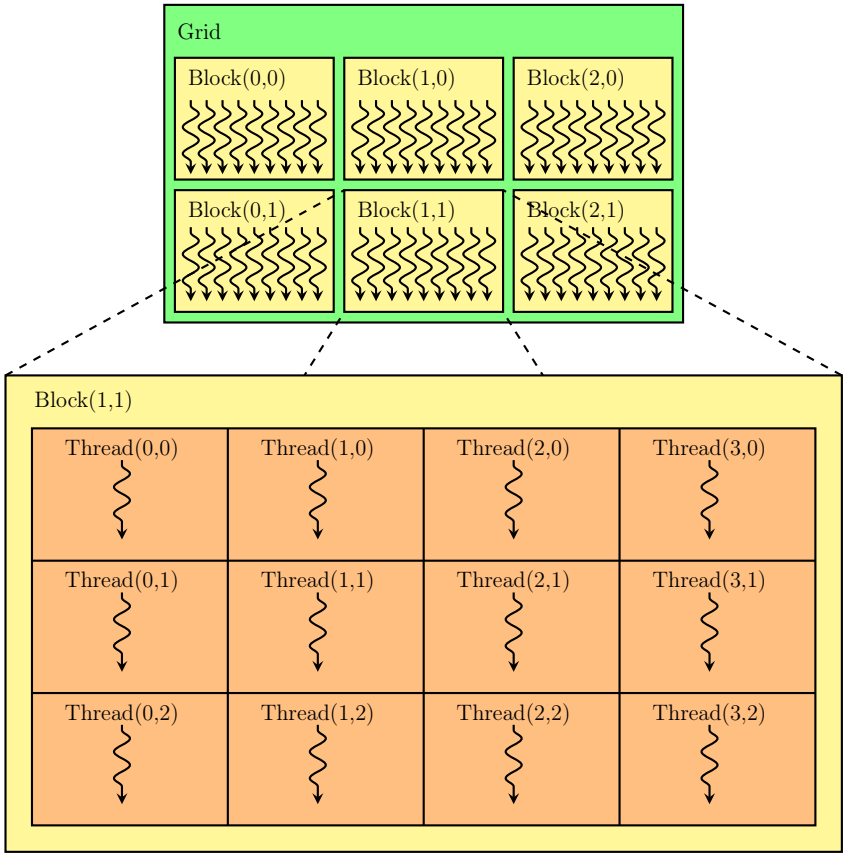

Figure 6: How the warp scheduler partitions the warps

Warps execute one instruction at a time, so maximum efficiency is only achieved when all threads go down the same branch. If the threads diverge on different branch paths, the scheduler will pause and run all branches taken serially. Only after all paths are run serially will the threads converge to the same execution path and continue in parallel. There is however only branch divergence within each warp, all warps run independently of each other regardless of path taken.

2.7.3 CUDA Cores

CUDA Cores are contained in each SM, how many CUDA Cores reside in an SM depends on the type of architecture. These CUDA Cores are used for integer and floating-point arithmetic operations. The contents of a CUDA Core are shown below in *Figure 7*.

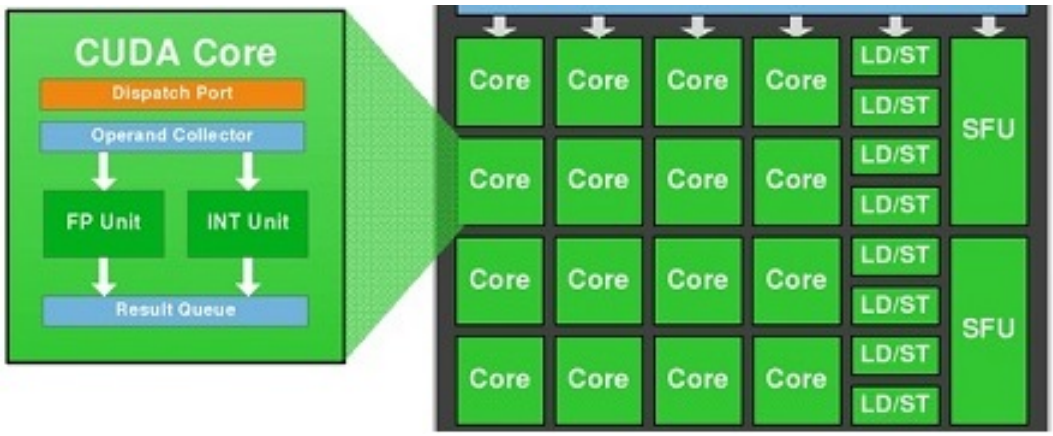

Figure 7: The insides of a CUDA Core

2.7.4 Programming GPUs with CUDA

CUDA utilizes *kernel functions* which consists of the computational tasks to be performed on the GPU. An application can consist of one or more kernels, and is written in either C/C++ or Fortran. Once a kernel is compiled, it makes use of many threads that execute the same code block in parallel. Multiple threads are grouped into thread blocks where all threads in a thread block run on a single SM. Within each thread block, threads cooperate and share memory. Each thread block is also divided into warps of 32 threads, where the warps are the fundamental units of dispatch within an SM.
An example of a CUDA program can be as follows:

```
1 // Kernel that executes on the CUDA device
2 __global__ void square_array(float *a, int N) {
3   // 1D thread blocks and 1D thread array inside each
      block
4   // N is the length of the data array a, stored in
      device memory
5
6   int idx = blockIdx.x * blockDim.x + threadIdx.x;
7   if (idx<N) a[idx] = a[idx] * a[idx];
8 }
```

sample_kernel.c

3 Methods

In this section I will talk about the different methods and algorithms needed to implement the paired-end merger. I will also discuss how this can be optimized for parallelisation on GPUs.

3.1 Hardware

For development and testing a machine issued by UiO named *samsida* was used. Its specifications are as follows:

- CPU: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (8 cores)
- GPU: Nvidia Quadro P6000 24 GB $x4$
- RAM: 125 GB

The Nvidia Quadro P6000 uses the GP102GL architecture, features 3840 CUDA Cores, and has a theoretical memory bandwidth of up to 432 GB/s.

3.2 Software

Samsida runs on a desktop version of Red Hat Enterprise Linux Server release 7.5 (Maipo) with a kernel version of 3.10.0-862.el7.x86_64. The system uses Nvidia Driver Version 390.48. For compiling CUDA code *nvcc* version 9.0.176 is used, the rest of the code is handled by *gcc* version 4.8.5. Profiling tools such as *gprof* version 2.27-28.base.el7_5.1, NVIDIA Visual Profiler and *nvprof* were used. The Visual Profiler and *nvprof* are tools which are included in the CUDA Toolkit. For comparing speed and correctness of the new algorithm *FLASH* v1.2.11 was used.
To generate Illumina reads for testing purposes a read simulator called ART Version 2.5.8 was used. This read simulator uses an existing genome as input to generate new reads. The read simulator can be downloaded from the link below:
`https://www.niehs.nih.gov/research/resources/software/biostatistics/art/index.cfm`

3.3 Data

Apart from the aforementioned generated reads made by ART, two additional read sets were used to compare speed and correctness. Both read sets were provided by GAGE-B, one set assembled from HiSeq Illumina reads, and another from MiSeq Illumina reads.
The genome assembled from HiSeq Illumina reads were *Bacillus cereus VD 118*, and from MiSeq Illumina reads *Bacillus cereus ATCC 10987*. The MiSeq genome contains 1040000 long read-pairs of about 251bp, while the HiSeq genome contains 6500000 short read-pairs of about 101bp.
The reads can be downloaded from
`https://ccb.jhu.edu/gage_b/datasets/index.html`

3.4 Optimization

A sequential solution to this problem would merge one pair then move on to the next, doing one merge at a time. Parallelisation is done on a code level in the software, which allows for different processes to simultaneously work on different reads. This is done to speed up the whole process of aligning reads and hopefully allow to make more precise calculations to get alignments that are true. Parallelisation is usually done on cores in CPUs, where each core get one or more tasks. The more cores the CPU has, the more tasks it can do at once. Parallelisation can also be done on GPUs, e.g. NVIDIA Tesla GPUs feature CUDA cores, these cores can also do calculations individually at the same time increasing speed.

When merging paired-end reads, the reads need to be kept in memory. The more reads you want to process at once, the more memory is needed. This also affects the speed of the program as these reads need to be transferred between the host and the device. When doing parallel programming on GPUs you need to take into account the amount of transfers done as this is one of the most time consuming operations. To achieve an optimized program the transfers must be at a minimum, allocating the necessary memory on GPUs first then transferring over the reads in batches before merging.

GPU parallelization is good for specific problems, e.g. matrix multiplication. In FLASH, the part which somewhat fits the nature of GPU parallelization is the *find_mismatches* function. This function finds the number of mismatches in a given alignment, which is then used to find where the two reads overlap.

The loop in *find_mismatches* which may benefit from GPU parallelization.

```
1 for(int i = 0; i < overlap_len; i++) {
2   if(seq1[i] != seq2[i]) {
3     num_mismatches++;
4     total_mismatch_qual += min(qual1[i], qual2[i]);
5   }
6 }
```

find_mismatch.c

This loop can either be run with different reads by different processes, finding multiple *num_mismatches* at once, or several processes can split the work on two reads. The first option is most likely to give the most speedup as the loop is relatively short, which gives quick runs by each process, allowing for more processes at once. If we were to split the processes for each two reads, we would need to tally up the results from each of the processes before we get the result needed, leading to more time spent.

3.4.1 Storing of the reads

How we store data is important to ensure that we get the speeds we want when doing the merging. In the initial software I stored reads in structs, one struct for forward-reads, and one for reverse-reads. This did not prove to be very effective as the GPU had to jump back and forth between a forward and reverse read which were not contiguous in memory. To solve this problem I chose to store both the forward and the accompanying reverse read in one struct so that they would be contiguous in memory. I also added the final merged read in the same struct for simplicity.

3.4.2 First iteration of the GPU code

In the first working parallel code I went with the approach of different reads for different processes to fully utilize the number of threads per SM. As the NVIDIA Quadro P6000 has 1024 threads per SM, 1024 seems a good number to divide the number of reads into. This ensures that 1024 threads work on a block of reads at once, hopefully maximizing the utilization of the GPU. In the code seen below we first prefetch the *overlap_starts* and the data contained in *reads_gpu*. When prefetching we tell the program to move the data to the GPU after it is initialized. This eliminates the need for the program to check whether the data is located on the CPU or the GPU, and then transfer it over. By doing this all the data is moved at once and we remove unnecessary page faults which only adds to time spent. After the prefetching is done, the kernel is called, and the *reads_gpu* and *overlap_starts* is sent to the GPU. Here multiple threads each work on a respective chunk of the *overlap_starts* array to fill it in.

```
1     // prefetch overlap_starts to GPU
2     int device = -1;
3     cudaGetDevice(&device);
4     cudaMemPrefetchAsync(overlap_starts,
          R_CHUNK*sizeof(int), device, NULL);
5
6     // Prefetching data in reads_gpu
7     cudaMemPrefetchAsync(reads_gpu->read_arr,
          R_CHUNK*sizeof(read), device, NULL);
8
9     int blockSize = 1024;
10    int numBlocks = (R_CHUNK + blockSize - 1) /
          blockSize;
11
12    combine_reads<<<numBlocks,
          blockSize>>>(reads_gpu, overlap_starts);
13
```

```
14      cudaDeviceSynchronize();
```

first_iter.c

3.4.3 Output of NVIDIA Visual Profiler

On the left side of the profiler we see the different categories which the program is divided into, in this case the main process, and the GPU.
In the main process we see the main thread which consists of the Runtime API and the Driver API, and the Profiling Overhead. The Runtime API consists of different API calls in the CUDA library, such as *cudaMallocManaged(), and cudaMemPrefetchAsync()*. The Profiling Overhead consists of the time the profiler spends on starting and stopping the profiling.
The Unified Memory category consists of CPU page faults, this is page faults happening on the CPU due to moving the data between the CPU and the GPU. The CPU is usually very quick to handle a CPU page fault so these are not of great concern.
The Quadro P6000 category is the main focus in the profiler. This category helps explain how much time is spent on data migration from host to device(HtoD) and device to host(DtoH), as well as GPU page faults. Generally when programming in CUDA we want to minimize the number of data migrations as well as page faults on the GPU as these are much slower to resolve than page faults on the CPU.
The Context 1 category shows how much time each kernel spends on calculation for each stream. In this program only one stream is utilized.
In the main window of the profiler we see the timeline. The bars we see here shows how much time is spent in each category, the longer the bar the longer the time spent.
On the bottom of the profiler we find the analysis tab which aids the programmer in finding what parts of the code or the kernel is not properly optimized or could be further optimized.
The bottom right is the Properties pane which tells details such as time spent in the selected category.

When running the first iteration of the code through the NVIDIA Visual Profiler we get the result shown in figure 8.

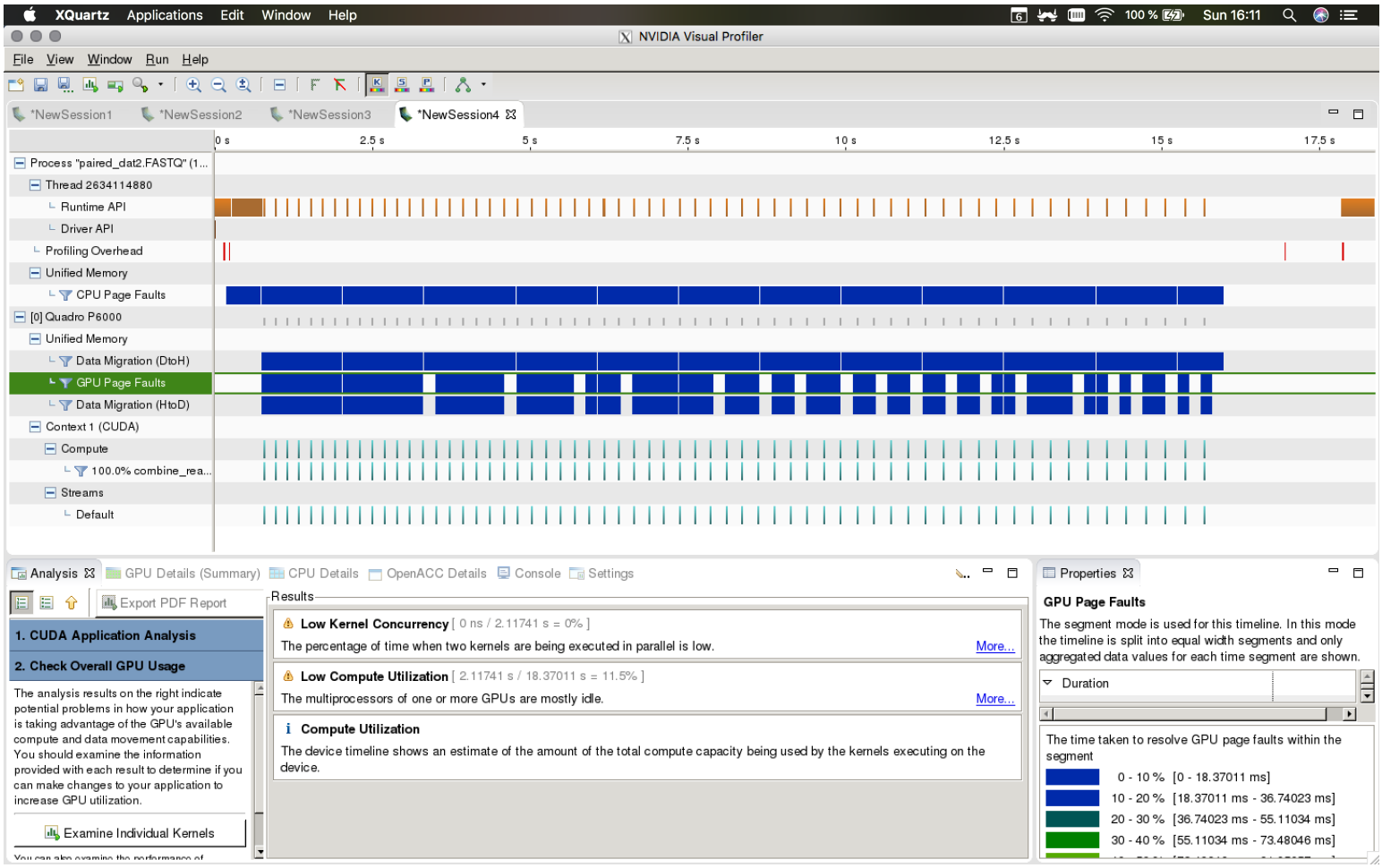

Figure 8: Result of first iteration in the NVIDIA Visual Profiler

Here we see Data migration between host and device, and Page Faults take most of the time. The kernel compute time is only a fraction compared to the rest. In this iteration the data is transferred over to the GPU in small batches one at the time, with many transfers between host and device. This in turn gives many kernel launches on small amounts of data, which is not optimal. This shows that either the storing or migration of data needs to be optimized if possible.

3.4.4 Second iteration of the GPU code

The first approach still causes many page faults and slow migration between device and host. To reduce migration time and page faults I decided to store the reads differently. A hopefully more efficient way would be to store every read in a contiguous array, and keep the index where each read start in a separate array. This means that only two arrays for the whole batch of forward and reverse reads, along with two index arrays will be transferred over to the device instead of one struct per read.

A string in C is stored as a char array, where each char takes 1 byte. If we store N reads in each array and each read is b chars long, we end up with a char array taking $\frac{(N*b)}{1000000}MB$. Doing this we need two arrays for the forward and reverse reads and two arrays for the forward and reverse scores. We also need an additional array to keep track of where in the array each read starts, for this we need an integer array for the forward and reverse reads which is N long.

After the change in the code, we see in the visual profiler(*Figure 9*) that the data migration time has been reduced significantly. Now most of the time is taken by the CPU page faults at the start of the program, and the kernel. The CPU page faults happen when the data first needs to be accessed by the GPU, The gap in the picture is when the program is finished with the current sequences and reads new sequences from the file.

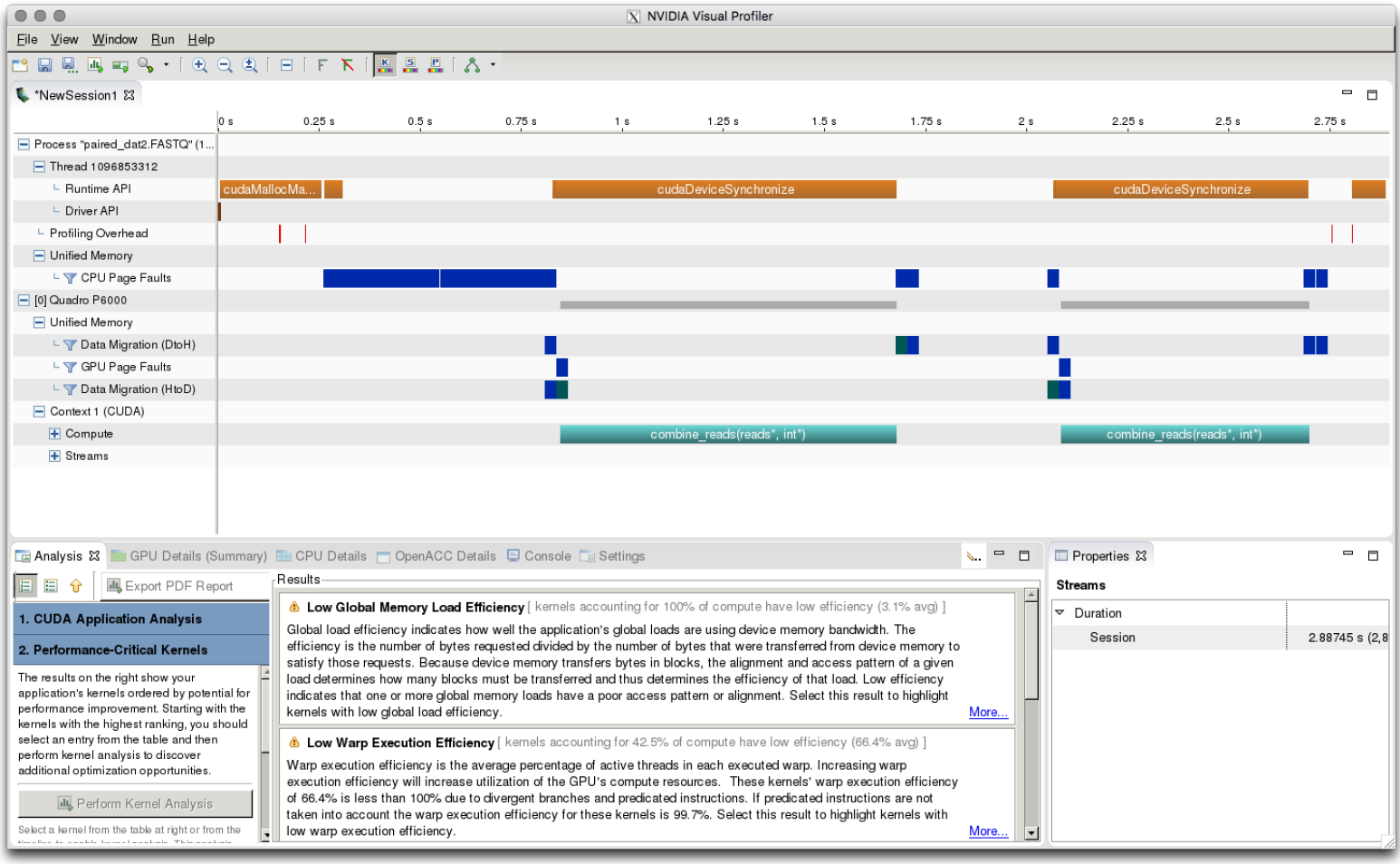

Figure 9: Result of second iteration in the NVIDIA Visual Profiler

With a small dataset of 200 MB the previous code used about 17 seconds to finish, with the current optimized storing and transferring of reads this takes about 9 seconds. In comparison the FLASH program takes about 1,7 seconds on the same dataset. With such a small dataset this program shows no improvement in terms of time, but with bigger datasets this will hopefully change. When it comes to correctness, the results show the same as FLASH, it merges the same amount of reads with the same overlap, and it gives the same non-merged reads. When it comes to scoring, there is obviously some difference since these are calculated differently than in FLASH.

Table 2: Snippet of precomputed scoring table

	...	**A**	**B**	**C**	**D**	**E**	**...**
...							
A		65.0	66.0	67.0	68.0	69.0	
B		66.0	67.0	68.0	70.0	71.0	
C		67.0	68.0	70.0	71.0	73.0	
D		68.0	70.0	71.0	73.0	73.0	
E		69.0	71.0	73.0	73.0	73.0	
...							

3.4.5 Speeding up merging

When doing the merging we are calculating the scores in accordance to the formulas given by Robert C. et al. 2015 as shown previously. Calculating these scores for every base can be quite time consuming. The formulas always gives the same output for two given scores. Knowing this, we can precompute all the possible outcomes.

Since there are 41 possible scores with each their own symbol, a matrix of 41^2 is necessary. Two of these matrices are needed since there are two different formulas, one formula for when the two bases are the same, and one for when the two bases are different. After these two tables are computed, we only need to do an array lookup with the two scores to find the new score.

The way the score was calculated previously:

```
1 double px = qual_to_p((int)*read_1_qual);
2 double py = qual_to_p((int)*read_2_qual);
3
4 double p = px * py / 3.0 / (1.0 - px - py + 4.0 * px
    * py / 3.0);
5 double merge_qual_same = opt_fastq_ascii +
    min(round(-10.0*log10(p)), opt_fastq_qmaxout);
```

old_score.c

Now replaced by a single line:

```
1 double merge_qual_same =
    same_qual[(int)*scores_f-33][(int)*scores_r-33];
```

new_score.c

After implementing the precomputed table, the time spent on the 200 MB dataset reduces to about 3 seconds. It is still not as fast as FLASH, but shows a considerable improvement over the previous iteration by more than halving the time. Implementing this table lookup causes no change in correctness of the merged reads, and merges the same amount of reads.

3.5 Implementation

What follows is detailed explanations of the finished implementation of the paired-end read merger.

3.5.1 Workload

When doing GPU programming we want most of the work to be done on the GPU, this however, proved difficult as this algorithm does not quite fit the nature of parallel programming on GPUs. I still transferred as much work as possible over to the GPU, while some parts of the implementation are left to the CPU. The process of finding the overlap between the two read-pairs had the possibility of gaining most from the GPU as this contains simple calculations which can be spread over multiple threads. The merging of the pairs on the other hand, best fit the CPU as this process contains no calculations, but instead only does table lookups and pointer copies. The overall process then becomes as follows:

1. CPU reads forward and reverse reads from file
2. CPU copies reads over to GPU
3. GPU finds the best overlap for each read-pair
4. CPU merges the read-pairs by the overlap returned by GPU
5. CPU writes final merged sequences to file

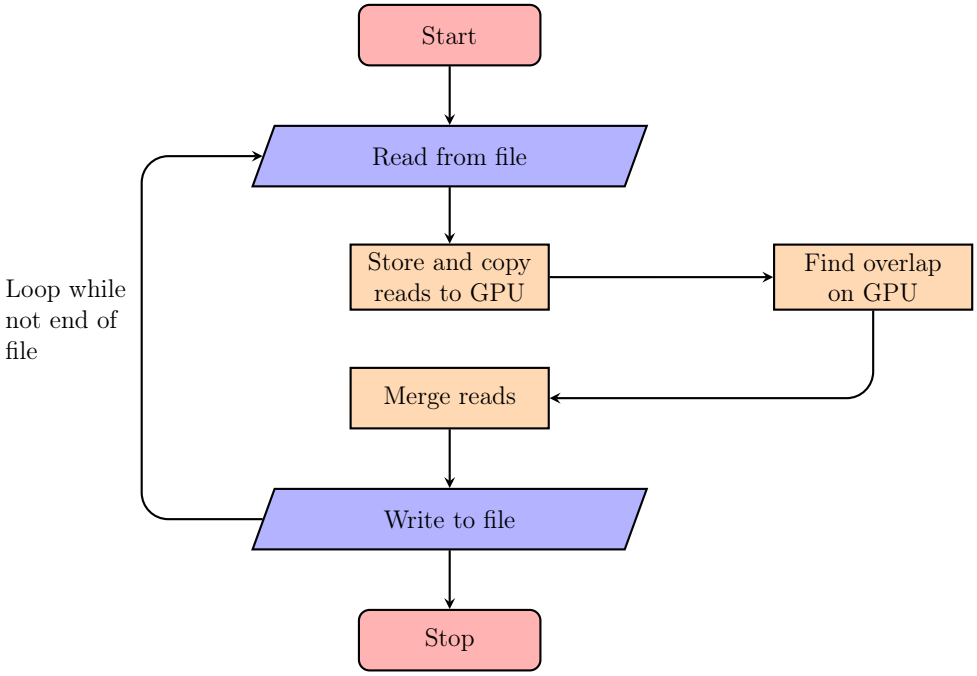

Figure 10: Flowchart of overall process

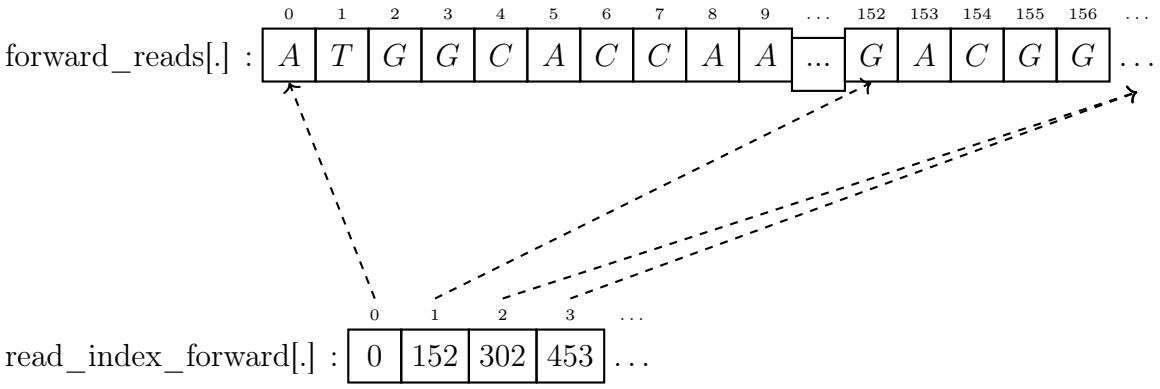

Figure 11: How the index references the reads in memory

3.5.2 Storing of reads

Before storing the reads, the program goes through the input file once to find the maximum length of the reads. The reason for this is so we know how much memory to allocate on the GPU before making the data accessible. Since Unified Memory is used in this implementation, it is unnecessary to do two allocations, one for the CPU and one for the GPU. Once the data is allocated with Unified Memory, anything copied to the allocated memory space will immediately be available to both the CPU and the GPU.
The file is read in chunks, and after each line it is copied to its respective allocated array, making it available on the GPU. When the reverse read is read it is reverse complimented along with its score before it is copied to the GPU. When the whole chunk is read and the Unified Memory arrays filled, the paired-end merger moves on to the next step, which is to find the overlap of the read-pairs on the GPU. The reads are represented as shown in Figure 11.

3.5.3 Kernel

```
1 __global__
2 void combine_reads(struct reads *reads_gpu, int
    *overlap_starts) {
3
4     int index  = blockIdx.x * blockDim.x +
        threadIdx.x;
5     int stride = blockDim.x * gridDim.x;
6
7     int num_reads = R_CHUNK;
8
9     // initialize data on GPU
10    for(int i = index; i < num_reads; i += stride) {
11        overlap_starts[i] = -1;
12    }
13
14    for(int i = index; i < num_reads; i += stride) {
15
```

```
16          int read_len_f =
                reads_gpu->read_len_forward[i];
17          int read_len_r =
                reads_gpu->read_len_reverse[i];
18
19          // align the pairs to find the where the
                overlap starts
20          overlap_starts[i] =
                align_pair(reads_gpu->reads_forward +
                i*read_len_f, reads_gpu->reads_reverse +
                i*read_len_r, reads_gpu->scores_forward +
                i*read_len_f, reads_gpu->scores_reverse +
                i*read_len_r, read_len_f, read_len_r);
21      }
22 }
```

kernel.c

The kernel shown above is run once for every chunk of reads. It first initializes the data in the *overlap_starts* array. Doing this, we eliminate any possible GPU page faults, since we first access it on the GPU. We then start the main loop which splits the calculation of overlaps on the different threads. How many threads per overlap depends on the size of the chunk of reads, and the length of the reads.

I decided to have a *blockSize* of 512 which is half of the max number of threads the Quadro P6000 supports in one threadblock. The reason for the big *blockSize* is to try to make up for the branching code in the *align_pair* function. Furthermore, of all the sizes tested, 512 seemed to be the best with a small margin.

The number of blocks is calculated as $(arr\_len\_f + blockSize - 1)/blockSize$, where arr_len_f is the total length of the array containing the chunk of reads. This seems to give an even distribution of the threadblocks, giving a good result.

3.5.4 Finding overlaps

When it comes to finding overlaps on the GPU, the code is the same as in the serial implementation, apart from some *__syncthreads()* functions to avoid race conditions between the threads in each warp. It is very similar to FLASH apart from the SIMD instructions which is absent in this implementation. The reason for not changing much here is because there were no easy ways to convert it to GPU code. If I were to run the kernel once per read I could utilize shared memory, as I could fit the read-pairs as well as their scores there. Doing this the kernel in itself would most likely be very fast, but since the process of finding the overlap is not very demanding the overhead for starting all the kernels would take longer than finding the overlaps. Furthermore, adding shared memory to the implementation as it is now would not be possible since it would need to store the whole batch of reads in shared memory, far surpassing the size of the GPU's shared memory capacity.

3.6 Differences from FLASH

Overall this implementation of read merging is quite similar to FLASH. FLASH however stores each read in a struct, and sends smaller batches to combining. Looking more closely at FLASH's inner workings, we see that it employs one reader thread, one writer thread, as well as several combining threads to minimize the amount of calculation downtime. In addition to using threads to speed up the process, it also utilizes SIMD instructions, more specifically SSE2 intrinsics. SSE2 makes use of the CPU's ability to do multiple calculations at once without threads, but instead placing data in registers, called vectors. FLASH uses SSE2 on the inner loop that counts the number of mismatches and calculates mismatch quality. This adds another level of parallelization to the task, and gives a speedup of about $2x$ according to the makers of FLASH.

4 Results and Discussion

4.1 Simulated datasets

In this section we look at the results from the generated read sets made by ART. ART needs a real genome to generate the reads, and the genome used here is the E.coli-genome (strain K-12 MG1655) with accession U00096.3. When running the GPU implementation and FLASH, the default parameters where used for both. The programs were run with a *max_ overlap* of 65, *min_ overlap* of 10, and *max_ mismatch_ density* of 0.25.

4.1.1 Runtime

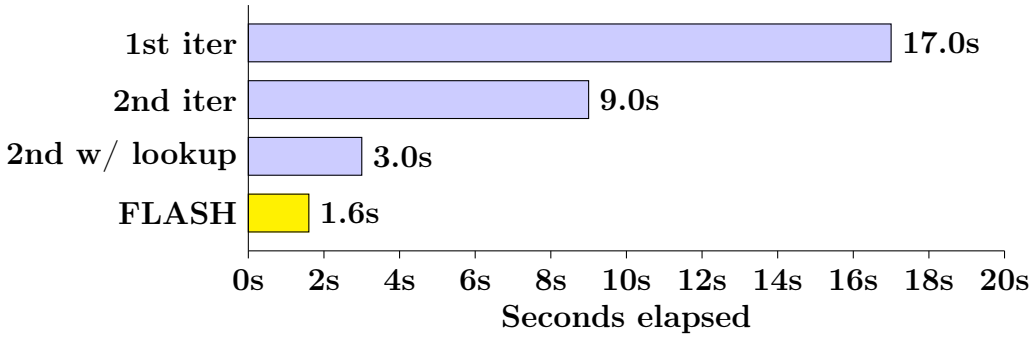

Figure 12: Runtime differences between the different implementations and FLASH on a generated 200 MB dataset

Comparing the three iterations shown in *Figure 12* on the 200MB dataset, we see it gradually gets faster. From the first to the second iteration it is sped up by 47%. This is mainly caused by the change in the storing of the reads, allowing more efficient access of data by the GPU and transferring a bigger amount of reads at once. This in turn gives the kernel more to work on, reducing the amount of overhead from starting many kernels in a short time span.

By adding the score lookup in the serial part of the code, we see a 66% speed up. The cause of this is eliminating all the expensive mathematical operations done per base of each read when merging. Instead of doing these expensive calculations, it calls the lookup function and finds the score in the precomputed table.

Comparing the final iteration with FLASH however, shows that FLASH is about 53% faster. This might be due to the small file sizes as GPUs spend some time on overhead when initializing the Driver APIs and allocating the memory beforehand. This means that the FLASH algorithm will have a big head start before the GPU implementation has even started. This will hopefully get solved with bigger file sizes, as this overhead is constant no matter the file size.

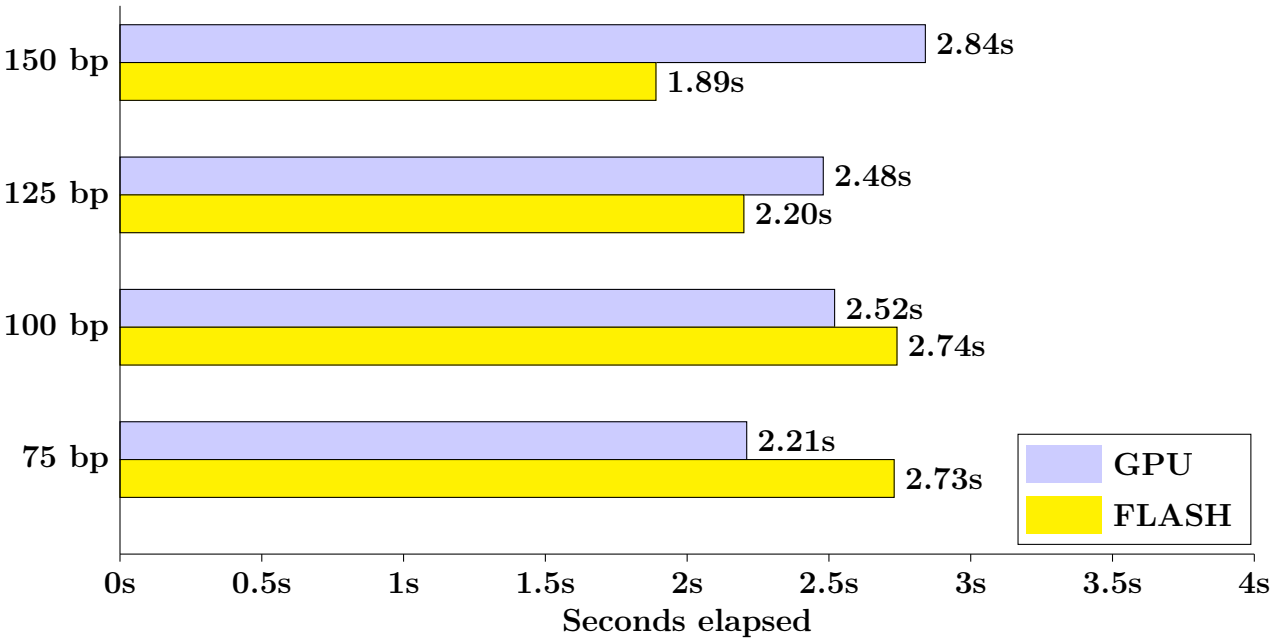

Figure 13: Runtime difference between final GPU implementation and FLASH with different read lengths

On this small dataset of varying read lengths we see that there are small differences between the GPU implementation and FLASH(*Figure 13*). While FLASH has a shorter runtime for the 150 bp and 125 bp, the GPU implementation is slightly faster at the two smaller read lengths. The reason for the slightly faster runtime on the GPU implementation on the shorter read lengths could be smaller amounts of branching in the GPU code. Another reason could be the extra output files and results FLASH produces compared to the GPU implementations which would increase time after all the read-pairs are merged.

4.1.2 Accuracy

The reads to measure the accuracy of the program is generated by ART, here I chose to have a fragment length of 350, coverage of 10, and 20 as standard deviation. I generated 3 read sets with a read length of 150, 200 and 250 bp. With these parameters, the 150 bp reads are supposed to have very few overlapping pairs whereas almost all of the 250 bp read-pairs should be merged.

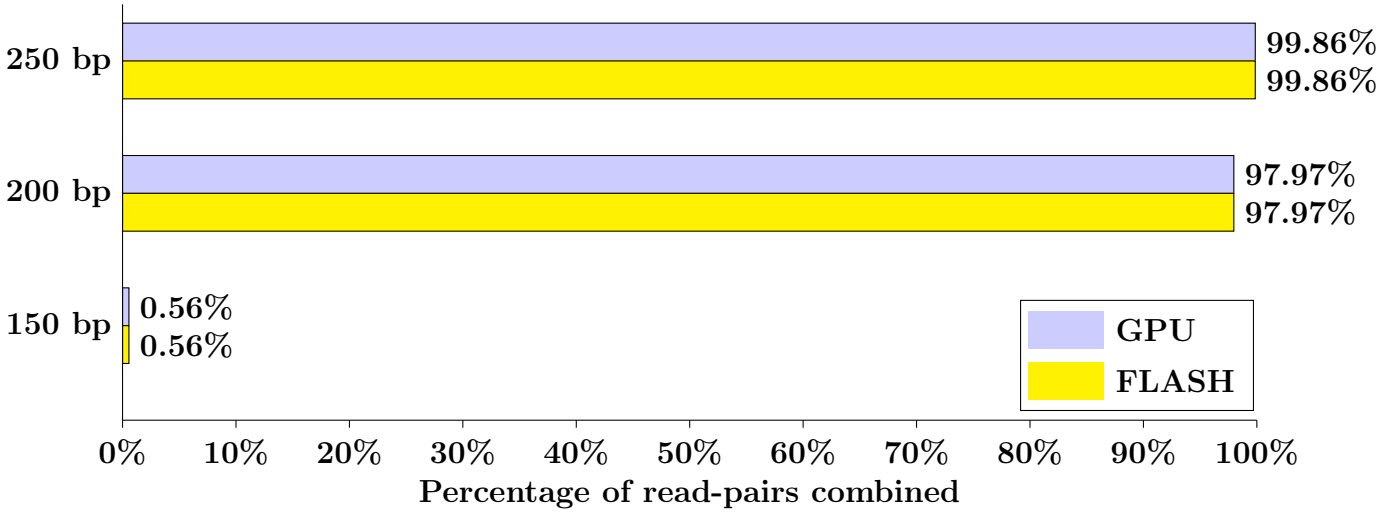

Figure 14: Percentage of combined reads in FLASH and the GPU implementation

Here we see that the two implementations merge exactly the same amount of reads regardless of read length. This shows that the correctness of the GPU implementation is on par with FLASH regarding simulated read-sets. There might be edge-cases not tested which produce different results. This would however most likely be a fault in the programming as FLASH's algorithm and the GPU implementation is the same and should produce the same result.

```
@U00096.3-139068
ATGGTGACAAAAGCCTGAAGCGCTACGCTTATCAAGCCTACAGATTCTCGCGCC
ACTCGTAGGCCGGATAAGGCGTTCACGCCGCATCCGGCAGTGGTGACAAATGCC
TGATGCGCTACGCTTATCAGGCCTACAGATTCTTGCGCCATTCGTAGGCCGGAT
AAGGCGTTCACGCTGCATCCGGAATGAACAATTAGGTCATACGCGGTAACGTTC
+
GF#DEGGG39G#EFG#CG#FGG+EGFEGGGGGGG#FG8FGGGDGDFGGFGFGGF
D@F?GGGEGGDCGGGGCGGGGF#GGGGGGGGFGGGGGGF,CGGGG#GGGGGGFG
GGGGG<GGGGGGFGCGGGGGGGGGGGGGGGGEGGGGGGGGF7G8GGCGG,>:+
GG*GGGF,F@+GG*GGG7;6#F#*G9GGG0##9+0C)G,#06CF#9G.#ABCG+

@U00096.3-139068
ATGGTGACAAAAGCCTGAAGCGCTACGCTTATCAAGCCTACAGATTCTCGCGCC
ACTCGTAGGCCGGATAAGGCGTTCACGCCGCATCCGGCAGTGGTGACAAATGCC
TGATGCGCTACGCTTATCAGGCCTACAGATTCTTGCGCCATTCGTAGGCCGGAT
AAGGCGTTCACGCTGCATCCGGAATGAACAATTAGGTCATACGCGGTAACGTTC
+
GF#DEGGG39G#EFG#CG#FGG+EGFEGGGGGGG#FG8FGGGDGDFGGFGFGGF
D@F?GGGEGGDCIIIGCIIGGIGIIIIIIIIFGIGIIIIGFIIIIGIIIIIIFI
IIIIIAIIIGGIFIFIIGGIIIIIIIIIIGIGIIGIIIIGGI7G8GGCGG,>:+
GG*GGGF,F@+GG*GGG7;6#F#*G9GGG0##9+0C)G,#06CF#9G.#ABCG+
```

Figure 15: Example of combined pairs from 150 bp run on FLASH and GPU implementation respectively

Figure 15 above shows two combined read-pairs, one from FLASH and one from the GPU implementation. We see here that the two read sequences produced are exactly the same, meaning that both implementations found the overlap at the same place. The score however, is slightly different where the two pairs overlap. The cause of this is the difference in how the two implementations calculate the score of bases which overlap.

4.2 Real datasets

The datasets used in this section are the read-sets from GAGE-B, one 1.5 GB dataset with long reads of 251 bp, and one 3.5 GB dataset with short reads of 101 bp.
Both programs were again run with default parameters as stated above in *Section 4.1*.

4.2.1 Runtime

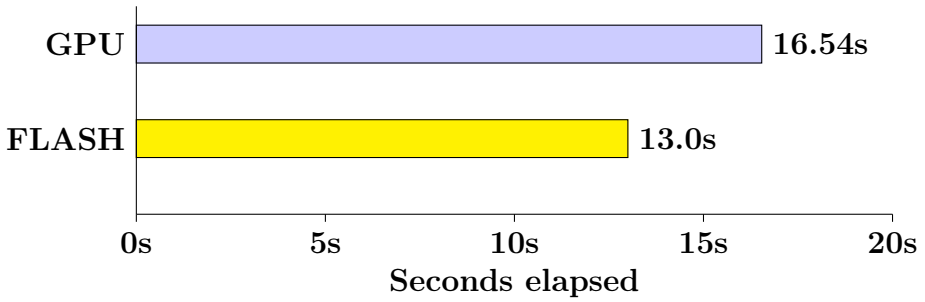

Figure 16: Runtime difference between FLASH and final implementation on a 1.5 GB dataset of long reads

As shown in *Figure 16*, we see the GPU implementation taking a bit longer than FLASH. The reason for this is most likely the nature of the algorithm used to find the overlaps, as this consists of big nested loops with several branches. The branches here are the main problem; each time a thread reaches the end of a branch it has to wait for all the other threads. This causes idling in many of the threads and an increase in time since fewer threads are working on the calculations.
The speed of the GPU implementation is however suspiciously close to FLASH when comparing it to other runtime results.

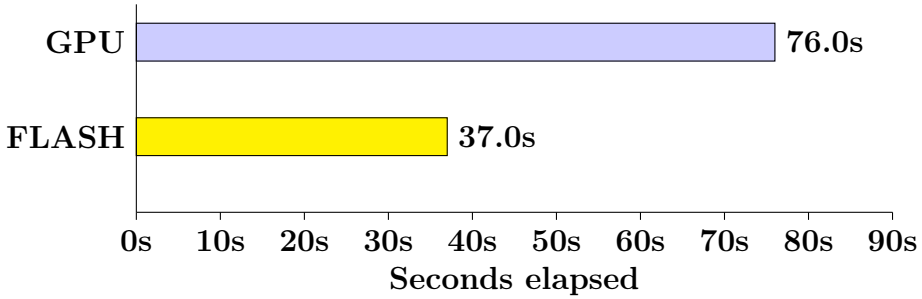

Figure 17: Runtime difference between FLASH and final implementation on a 3.5 GB dataset

On the 3.5 GB dataset we see that FLASH is much faster than the GPU implementation. This dataset produces less branching due to shorter reads than the previous dataset, but the GPU implementation still takes significantly longer. The branching and the time spent in the kernel is still a significant portion of the time spent, but time also adds up during transferring data between host and device. Since FLASH does not need to transfer data between host and device, it saves some time compared to the GPU implementation. The result is shown in *Figure 17*.

4.2.2 Combine time

In this section, I will look at how long the program spends in the different parts of the program, splitting up the combining/merging and other parts considered as overhead. To get these results, I timed the different methods in the GPU implementations with the C library function *clock_t clock(void)* and added them together. This however, again creates some overhead which may increase the total runtime, but should still give us a clear view of what parts in the program takes most of the time. The timing was was done on the 3.5 GB HiSeq dataset.

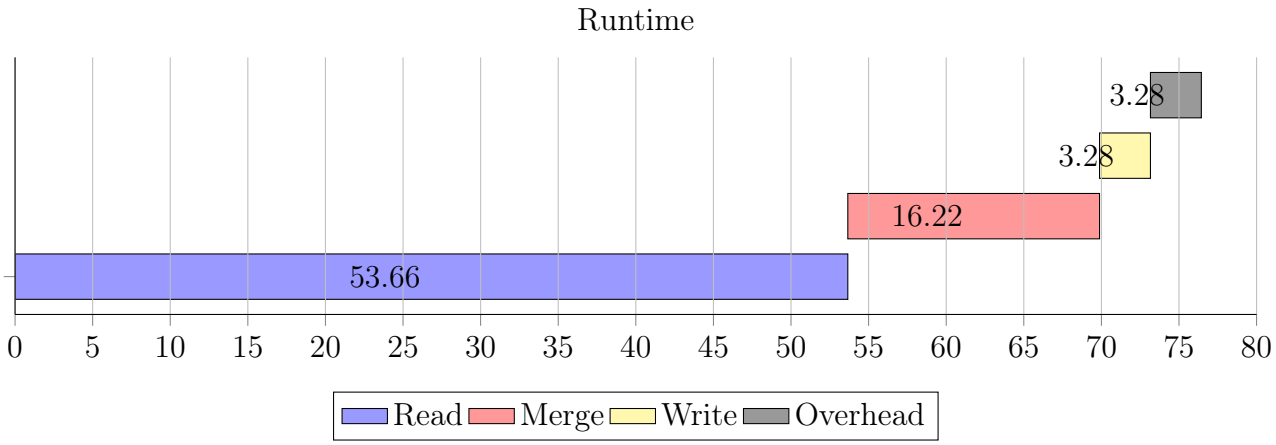

Figure 18: Runtime in seconds of the different parts in the GPU implementation

Figure 18 above shows how much time each of the parts in the implementation spends on the 3.5 GB dataset. Here, we see reading from the file takes most of the time, with 70.2%. Finding the overlap and merging only takes about 21.2% of the time. The overhead shown in the figure is the time spent on initializing CUDA, and allocating some data-structures at the beginning of the program. To try and find the cause of why the read function is taking so much time, we can try to split up the runtime in chunks, and time each chunk independently.

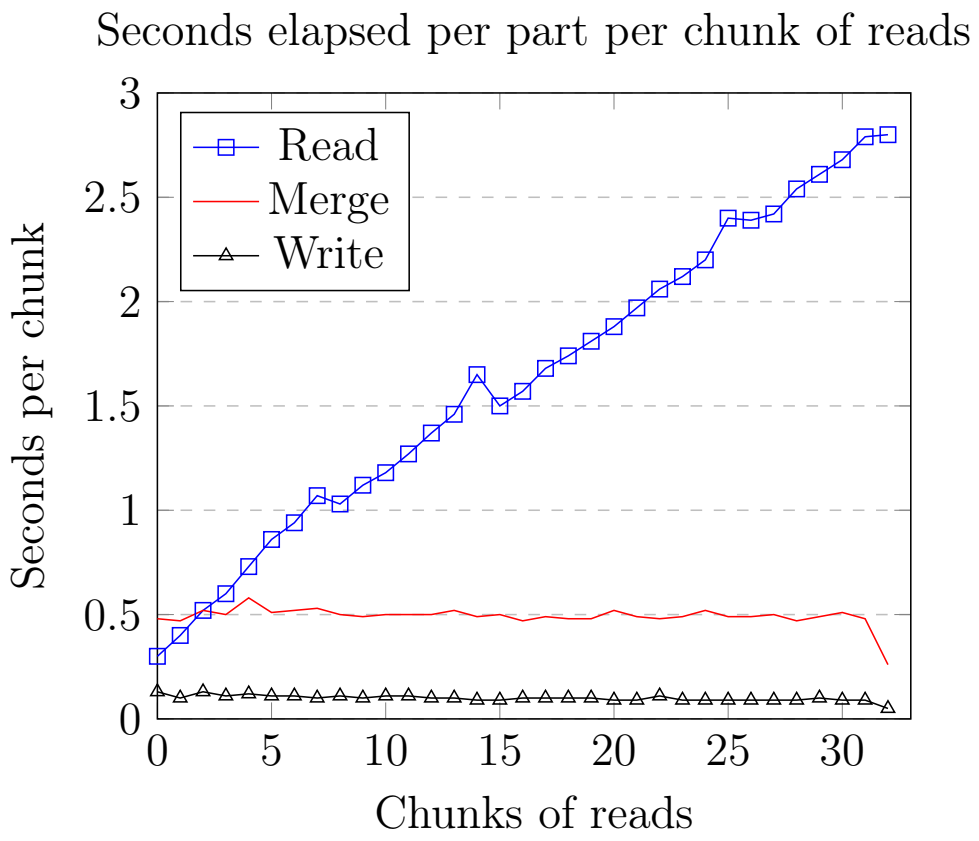

Figure 19: Plot showing how much time is spent in each of the parts of the implementation per read-chunk

The plot in *Figure 19* gives us a clear view of what takes time in the GPU implementation, and what needs to be optimized. Both the merging and the writing to file takes about the same time per chunk. The parts all have a slight dip in the last chunk since this chunk contains less reads than the previous ones. We see here that the function which reads from file increases in time for each chunk. This is because the reading from file is poorly optimized, and this was harder to spot on the smaller filesizes. The reason for the increase in time per chunk is a pointer which is used to keep track of where we left off in the previous chunk. When a chunk is finished and before a new one can be processed, this pointer must first process through the file to find the last position. This means that the file is read multiple times, whereas it should only be read once. An option for optimizing this will be discussed in *Section 6, Future work*.

4.2.3 Accuracy

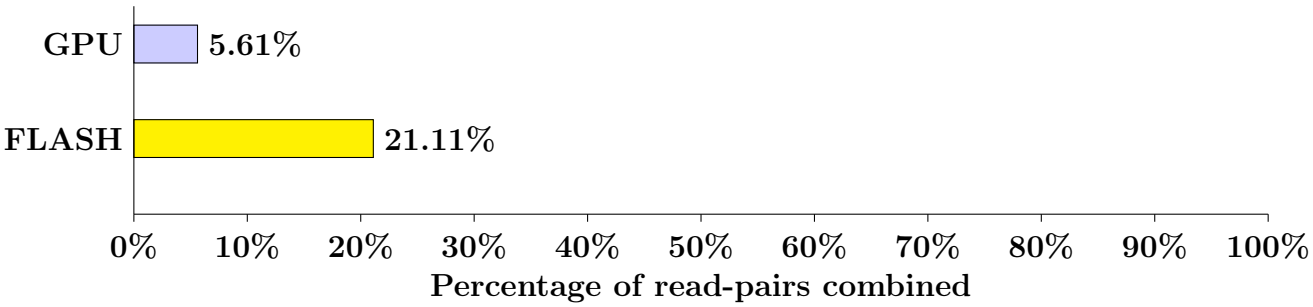

Figure 20: Percentage of combined reads in FLASH and the GPU implementation on the 1.5 GB GAGE-B read-set

As shown in *Figure 20*, the GPU implementation only combines about 6% of all the read-pairs whereas FLASH combines about 21%. From this we see that there definitely is an error in the GPU implementation. Further analysis done by inspecting the output files shows that the GPU implementation also incorrectly combines read-pairs that should not be combined, leading to a very different result compared to FLASH. The reason for the difference in the two implementations is hard to determine. Looking at both the code and the results shows no obvious errors, but this error also seems to affect the speed of the run. As shown in *Figure 16*, the actual runtime is close between the two, giving an outlier compared to the other runs with different datasets. This might indicate that the implementation for some reason skips part of the code or read-sets altogether, as skipping a big portion of the reads definitely will give shorter runtimes. The reason for the skips might be due to a combination of long reads with the addition of Ns.

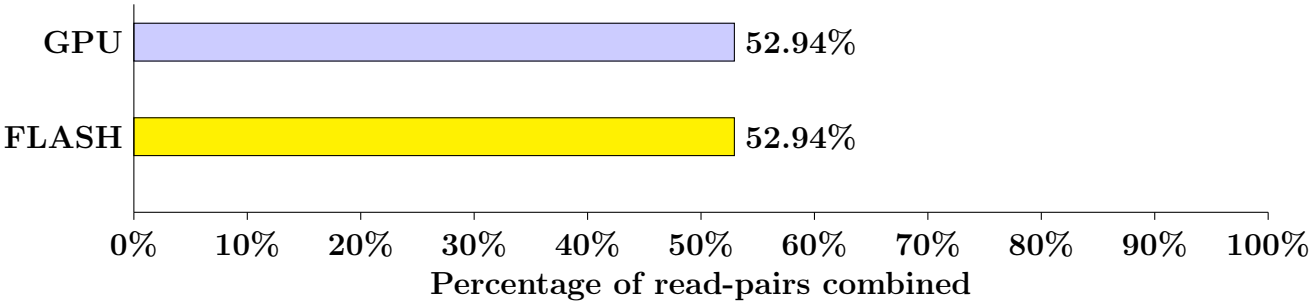

Figure 21: Percentage of combined reads in FLASH and the GPU implementation on the 3.5 GB GAGE-B read-set

Figure 21 above, shows that FLASH and the GPU implementation combines the same amount of read-pairs when it comes to the 3.5 GB dataset of short reads. Since this data-set has shorter reads and at the same time contains Ns, it further adds to the suspicion that the error produced by the 1.5 GB dataset might be due to a combination of Ns and the read-length. Also, the 250 bp run on the simulated read-sets combined the same amount of read-pairs as FLASH, but this read-set did not contain Ns. This shows that the implementation can handle long reads and reads with Ns separately, but maybe not both at the same time.

Figure 22 under shows an example of two reads from the 3.5 GB run of short reads. We see that the result from the combining produced an equal overlap, giving identical combined reads, but a slightly different score in the overlap which is to be expected.

```
@SRR497464.23 23
AGCTATAGCAATGAATTTAAGAAAGTTATCCACAAAATCAATACCTTGTG
GAATAAACTTGTCCACAATACGATATACTGTGTAAAAGTAAAAAAAGAGT
TTGTGGATACAAAAAAGAGAAAATATATTTATCCACA
+
@@<DDD>D?AFD?:?,22AE>DEGBHF<CHEFFFFDAGFIF@?FDDCCBC
CC@BBDAB?BD?CCEFFFF>FFC3@=.EHEGHDJIHGDGGIHF<IEFBDC
9F@F?C:C:AABHAHC<:<GEA2EDHFF8DD3BB8?:

@SRR497464.23 23
AGCTATAGCAATGAATTTAAGAAAGTTATCCACAAAATCAATACCTTGTG
GAATAAACTTGTCCACAATACGATATACTGTGTAAAAGTAAAAAAAGAGT
TTGTGGATACAAAAAAGAGAAAATATATTTATCCACA
+
@@<DDD>D?AFD?:?,22AE>DEGBHF<CHEFFFFDAIIIFAAFFICCBC
CCABBDABABDACCEFFHFAFFCAAAAEHEGHDIIHGDGGIHFAIEFBDC
AF@F?C:C:AABHAHC<:<GEA2EDHFF8DD3BB8?:
```

Figure 22: Example of combined pairs from 3.5 GB run on FLASH and GPU implementation respectively

4.3 Runtime analysis

Looking at both the simulated datasets and the real datasets, the speed of the GPU implementation is a bit behind when it comes to relevant read lengths. As shown in *Section 4.2.2* however, it should at least be possible to get the runtime much closer to FLASH on bigger files by changing how the program reads from file. With further optimization methods this could potentially be even faster than FLASH.

4.4 Accuracy analysis

Apart from the wrong result on the long-read 1.5 GB dataset, all the other tests showed that the accuracy of the GPU implementation was the exact same as the others. This however, is mainly due to the very few changes in the FLASH algorithm which finds the overlaps. If this method was to be further optimized and rewritten, it might have led to different results.

5 Future work

During the coding and writing of the thesis, I discovered several possible methods that could be used to optimize the GPU merger. Some options where already mentioned in the previous section which will be further explained along with other methods. The GPU implementation developed during this master thesis is more of a prototype, and to reach a fully functional and usable program at least some of the options mentioned in this section would need to be implemented. Some of the methods explained in this section, like the memory mapped files and CPU threads should not prove a problem to implement, but due to time restraints this was not possible.

5.1 General improvements

Although this master thesis is focused on GPUs there are still improvements that can be done on the serial parts of the implementation, both regarding speed and functions. The general improvements mentioned here contains improvements that could increase the speed of the program as well as improvements that would make the program more complete.

5.1.1 Memory mapped files

As stated in *Section 4.2.2* the current method to read from file in batches is poorly optimized. The method currently uses the C library function `FILE *fopen(const char *filename, const char *mode)` to open files, and the C library function `ssize_t getline(char **lineptr, size_t *n, FILE *stream)` to read one line at a time. These methods are not suited for bigger files which is very common when doing read merging. Memory mapped files could be used instead, as this can greatly increase speed when working with big files. `mmap()` is a system call that can create a virtual map for a whole file, instead of copying the file to RAM. We then get a mapped array that can instantly access whatever part we need in the file, eliminating having to search through the file line by line and increasing speed.
Doing this, it can be imagined that the speed of reading from file and copying to unified memory per batch can be reduced to a constant time of about

0.3 seconds as shown in the first batch in *Figure 19*. This could then potentially reduce the total time of reading from file on the 3.5 GB read-set from 53.66 seconds to about 10 seconds. The total runtime for the GPU implementation would then be reduced to about 33 seconds, slightly faster than FLASH.

5.1.2 Independent reader, combiner and writer threads

Due to time constraints, I did not have time to implement independent threads to handle concurrent reading, combining, and writing. This could give a great speed-up to the program as it would allow the GPU to continuously do work after the first batch is read, eliminating some of the need to wait both for the GPU and CPU. Adding this feature would also give a more "correct" analysis when comparing the GPU implementation and FLASH, as FLASH has this feature.

5.1.3 Allow for outies

A feature which FLASH supports is the merging of "outies". These are read-pairs that does not overlap in the middle, but toward the edges, an example of this is shown under in *Figure 23*. This is a feature which is not enabled by default, but must be enabled manually when running the program. If this feature is enabled, the program tries to find the best overlap in the "outie" region after the normal overlapping region, potentially almost doubling the runtime of the program since the overlap must be found twice.

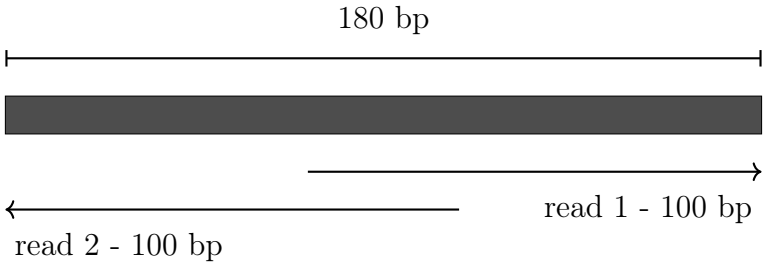

Figure 23: Representation of how outie reads are overlapped

5.1.4 Command line arguments

Currently the implementation only supports the input FASTQ files as arguments. Ideally, the implementation should support the important arguments such as *max_overlap*, *min_overlap* and *max_mismatch_density*. These variables must now be changed in the code to be modified. This is however a very easy feature to add, which I could not add again due to time constraints. Furthermore, the implementation could support the other arguments FLASH also supports, such as changing the phred offset to support different sequencing platforms, and setting the read- and fragment-length.

5.1.5 Read input

FLASH allows for interleaved input files. This means that it can accept both forward and reverse reads in one file, where the reads are interleaved. To implement this, the function which reads from file needs to be rewritten, or a new method needs to be added. Currently, the GPU implementation only supports standard FASTQ files with forward reads in one file, reverse reads in another, and a phred offset of 33.

5.1.6 Fixing accuracy errors

As shown *Section 4.2.3*, the implementation does not always produce the correct result. It seems that this is a flaw in the programming and most likely a simple mistake. It is however a necessary part to fix if this is to be a fully functionally program. With more time, I am certain that this is fixable, and would not be a deciding factor for whether paired-end merging could be done on GPUs.

5.2 GPU improvements

There is almost always room for optimization when it comes to parallelization, whether it is on CPUs or GPUs. The difficulty of optimizing however, gets gradually more difficult with each speed-up. There are definitely potential methods of speeding up this GPU implementation, some of them are listed in this section, but there are most likely many more.

5.2.1 Smaller kernel

One potential method of reducing time in the algorithm is to only run the inner loop which counts number of mismatches in a kernel and optimizing this for CUDA. To optimize this we can let each *threadIdx.x* handle one comparison between the bases and one *min()* operation and store the results in two arrays. After this we need to do reduction to add up all the results, doing reduction can be heavily optimized and sped up on GPUs.
However, with this approach, the kernel needs to run once per possible overlap for every read-pair. Looking at the kernel in a closed system it could be quite fast, but most of the time would be spent on overhead to start all the kernels, making this method unsuited for this task.

5.2.2 Concurrent kernels with streams

NVIDIA graphics processors with Compute Capability 2.0+ has the ability to employ concurrent kernels with the help of streams. According to Rennich 2012, a stream is defined as "A sequence of operations that execute in issue-order on the GPU". This means that each stream is a serial operation which can contain thread parallelization in the kernel, and these streams can run in parallel. In the current implementation the transferring of data to device, kernel-run, and transfer of data back to host is done serially, with parallelization only within the kernel. With streams it is possible to split up all three parts and run the different parts concurrently as shown in *Figure 24* under. In this figure the yellow bar is the copying from host to device, the green bar the kernel, and the blue bar the copying from device to host. With 4-way concurrency it is possible to achieve a $3x+$ performance improvement.

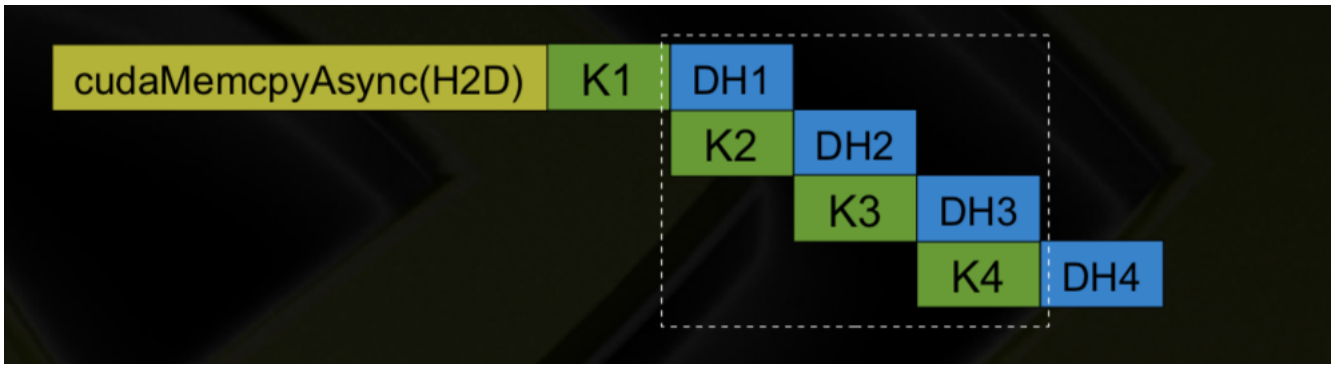

Figure 24: How 2-way concurrency looks

5.2.3 Rewrite of algorithm to find overlap

In this implementation not much is changed in comparison to FLASH in the algorithm to find overlaps. The algorithm in itself is not complex, but it does not quite suite the parallel nature of GPUs without a total rewrite due to the many branches. This rewrite however, would be quite extensive and time consuming to accomplish. This rewrite could theoretically utilize shared memory and possibly CUDA intrinsics to get better efficiency per kernel which could again lead to a speed-up.

5.2.4 Configuration of the GPU

At the moment, the batches which is read and copied to the GPU is a constant value of 200000. This value is selected as it fits the NVIDIA Quadro P6000 and the RAM on the *samsida* workstation, but if a different card or workstation were to be used, this value might cause the program to slow down, or even outright not work at all. Therefore, this chunk value would need to be calculated based on the specifications of the graphics card in the computer.

In addition, the number of threads per thread-block might also need to be changed if different graphics cards is used. Older graphics cards might not have the power to support 512 threads per block, and they might need to settle with 256 or 128 threads per thread-block to function.

5.2.5 Multiple GPUs

The *samsida* workstation provided by UiO has 4 of the NVIDIA Quadro P6000 graphics cards. It is possible to write the code such that the work is spread over all the 4 GPUs, but this would require heavy restructuring of the code and would become complex. Although it is four cards, it might not reach $4x$ performance when using all four compared to just one card, but it could still get a respectable speed-up.

6 Conclusion

This GPU implementation has proved that it is possible to use GPUs to do paired-end read merging on sequences produced by Illumina sequencers. The implementation has been optimized throughout the master thesis through several steps and has inched closer to competing tools. Although this implementation currently cannot quite compete with other existing tools such as FLASH, some of the easier further optimization methods mentioned could potentially make it faster than FLASH.

Through the results and discussion, I can say that even though paired-end read merging does not quite fit the nature of GPU parallelization, it is still fully possible to utilize parallelization on some parts of the algorithm. By dividing the work of finding the overlap on the GPU, the CPU can do the remaining tasks such as reading from and writing to file, and merging the reads which cannot be done on a GPU.

The mathematical formulas which replaced FLASH's score calculation also fit well in the implementation after being optimized. It did not clash with the GPU's more demanding work and neither hogged CPU time, stalling the rest of the program.

From this, I can conclude that merging of paired-end reads on GPUs definitely is a promising approach if optimized properly. With future graphical processors only increasing in performance, the time spent on merging will decrease if the sizes of read lengths or genomes does not increase. But with proper optimizing, longer read lengths and genomes might not even be a problem.

References

[1] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants". In: *Nucleic Acids Research* 38.6 (Dec. 2009), pp. 1767–1771. DOI: 10.1093/nar/gkp1137.

[2] Illumina. *Advantages of paired-end and single-read sequencing*. URL: https://www.illumina.com/science/technology/next-generation-sequencing/paired-end-vs-single-read-sequencing.html (visited on 07/28/2018).

[3] Illumina. *An introduction to Next-Generation Sequencing Technology*. 2017. URL: https://www.illumina.com/documents/products/illumina_sequencing_introduction.pdf (visited on 05/21/2017).

[4] Illumina. *FASTQ Files*. URL: https://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_FASTQFiles.htm (visited on 05/21/2017).

[5] Tanja Magoc and Steven L. Salzberg. "FLASH: fast length adjustment of short reads to improve genome assemblies". In: *Bioinformatics* 27.21 (Sept. 2011), pp. 2957–2963. DOI: 10.1093/bioinformatics/btr507.

[6] NVIDIA. *CUDA C Programming Guide*. 2012. URL: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (visited on 07/25/2018).

[7] NVIDIA. *NVIDIA Quadro P6000*. 2016. URL: https://images.nvidia.com/content/pdf/quadro/data-sheets/192152-NV-DS-Quadro-P6000-US-12Sept-NV-FNL-WEB.pdf (visited on 04/21/2018).

[8] Steve Rennich. *CUDA C/C++ Streams and Concurrency*. 2012. URL: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWe
pdf (visited on 07/30/2018).

[9] Edgar Robert C. and Henrik Flyvbjerg. “Error filtering, pair assembly and error correction for next-generation sequencing reads.” In: *Bioinformatics* 31.21 (2015), pp. 3476–3482. DOI: 10.1093/bioinformatics/btv401.